Hooks

¿qué es un Hook?

Hooks son funciones que te permiten "enganchar" el estado de React y el ciclo de vida desde componentes funcionales.

Los Hooks no funcionan dentro de las clases, te permiten usar React sin clases.

React proporciona algunos Hooks incorporados como useState. *

use Effect *

use Context *



useState

```
const [state, setState] = useState(initialState);
```

{ Devuelve un valor con estado y una función para actualizarlo. }

→ Durante el renderizado inicial, el estado devuelto (state) es el mismo que el valor pasado como primer argumento (initialState)

→ La función setState se usa para actualizar el estado. Acepta un nuevo valor de estado y sitúa en la cola una nueva renderización del componente.

```
setState(newState);
```

En las renderizaciones siguientes, el primer valor devuelto por useState será siempre el estado más reciente después de aplicar las actualizaciones.

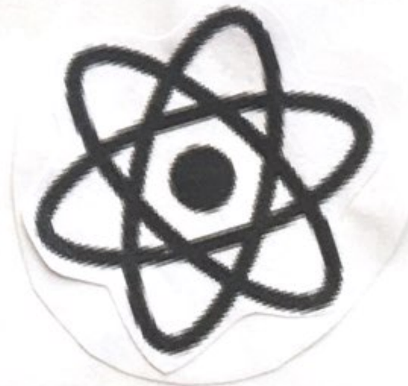
⚠ El argumento initialState es el estado utilizado durante el render inicial. En renderizados posteriores, se ignora.

con clases

```
class Contador extends React.Component {
  constructor (props) {
    super (props);
    this.state = {
        count : 0 };
  }

  render() {
    return (
    <div>
     <p> Hiciste click {this.state.count} veces </p>
  <button onClick={() => this.setState({ count :
   this.state.count + 1 })}> Haz click!
  </button>

  </div>
  );
 }
}
```



con hooks

```
import React, {useState} From 'react'
Function Contador () {
  const [count, setCount] = useState(0)
   <div>
    <p> Hiciste click {count} VECES </p>
   <button onClick={() => setCount(count + 1)}> Haz click!
   </button>
   </div>
  );
}
```



- Importamos useState que nos permite mantener un Estado local en un Componente Funcional.
- Dentro del componente Contador declaramos la variable count (guarda el número de clicks) llamando al hook useState. La Inicializamos en cero pasando (0) como único argumento a useState.

  SetCount nos permite actualizar el count.
- Cuando el usuario hace click, llamamos a setCount con un nuevo valor.

  React actualiza el componente Contador pasándole el nuevo valor de count.

useEffect

Al usar este Hook, le estamos indicando a React que el componente tiene que hacer algo después de renderizarse. React recordará la función que le hemos pasado (nos referimos a ella como nuestro "efecto") y la llamará más tarde después de actualizar el DOM.

useEffect equivale a la combinación de:

- componentDidMount,
- componentDidUpdate,
- componentWillUnmount.



con clases



```
class FriendStatusWithCounter extends React.Component {
  constructor(props) {
    super(props)
  this.state = { isOnline: null };
 this.handleStatusChange = this.handleStatusChange.bind(this);

componentDidMount() {
document.title = `Clickeaste ${this.state.count} VECES`;
chatAPI.subscribeToFriendStatus(
  this.props.friend.id,
  this.handleStatusChange); }

componentDidUpdate() {
document.title = `Clickeaste ${this.state.count} VECES`; }

componentWillUnmount() {
   ChatAPI.unsubscribeFromFriendStatus(
      this.props.friend.id,
      this.handleStatusChange ); }

 handleStatusChange(status) {
  this.setState({
     isOnline : status.isOnline }); )

 render() {
 if (this.state.isOnline === null) {
       return 'Loading...'; }
   return this.state.isOnline ? 'Online' : 'Offline'; }
 }
```

En el ejemplo anterior se añadió una funcionalidad: Actualizar el título del documento con un mensaje personalizado que incluye el número de clicks.

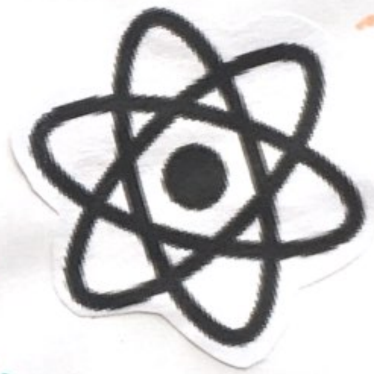
Esta lógica se divide entre componentDidMount y componentDidUpdate.

La lógica de la suscripción (tenemos un módulo ChatAPI que nos permite suscribirnos para saber si un amigo está conectado) se reparte entre componentDidMount y componentWillUnmount

SE SUSCRIBE

SE CANCELA LA SUSCRIPCIÓN





con hooks





¡Puedes usar el Hook de efecto más de una vez!

Esto nos permite separar la lógica que no está relacionada en diferentes efectos

```
Function FriendStatusWithCounter(props) {
const [count, setCount] = useState(0);
useEffect(() => {
   document.title = `Clickeaste ${count} veces`;
  });

 const [isOnline, setIsOnline] = useState(null);
 useEffect(() => {
  Function handleStatusChange(status) {
   setIsOnline(status.isOnline); }

 ChatAPI.subscribeToFriendStatus(props.friend.id,
 handleStatusChange);
  return () => {
  ChatAPI.unsuscribeToFriendStatus(props.friend.id,
  handleStatusChange); };
 });

 //...
}
```

useContext



```
const value = useContext(MyContext);
```

ACEPTA UN OBJETO DE CONTEXTO (el valor devuelto de React.createContext) Y DEVUELVE EL VALOR DE CONTEXTO ACTUAL.

EL VALOR ACTUAL DEL CONTEXTO ES DETERMINADO POR LA PROPIEDAD value DEL <MyContext.Provider> ASCENDENTEMENTE MÁS CERCANO EN EL ÁRBOL AL COMPONENTE QUE HACE LA LLAMADA.

> Cuando el <MyContext.Provider> ascendentemente más cercano en el árbol se actualiza, el Hook activa una renderización con el value más reciente del contexto pasado a ese proveedor <MyContext>

¡No olvides que el argumento enviado a useContext debe ser el objeto del contexto en sí mismo:

useContext(MyContext) !



hooks adicionales

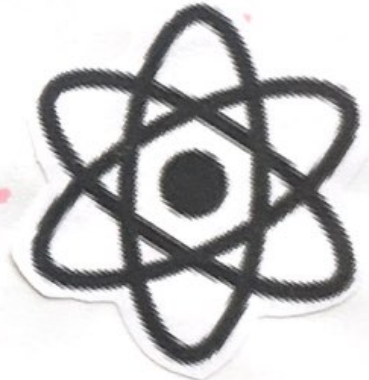
useReducer:

```
const [state, dispatch] = useReducer(reducer, initialArg, init);
```

Es una alternativa a useState. Acepta un reducer de tipo (state, action) => newState y devuelve el estado actual emparejado con un método dispatch.

useCallback:

```
const memoizedCallback = useCallback(
  () => {
    doSomething(a, b); },
  [a; b],
);
```



Pasa un callback en linea y un arreglo de dependencias. useCallback devolverá una versión memorizada del callback que solo cambia si una de las dependencias ha cambiado.

useMemo:

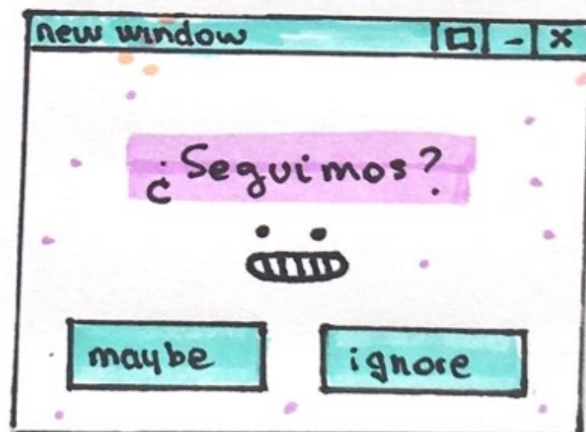
```
const memoizedValue = useMemo(() => computeExpensiveValue(a,b),
[a,b]);
```

Devuelve un valor memorizado.
Pasa una función de "crear" y un arreglo de dependencias.
useMemo sólo volverá a calcular el valor memorizado cuando una de las dependencias haya cambiado.
La función useMemo se ejecuta durante el renderizado.

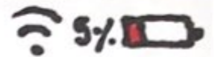
useRef:

```
const refContainer = useRef(initialValue);
```

useRef devuelve un objeto ref mutable cuya propiedad .current se inicializa con el argumento pasado (initialValue). El objeto devuelto se mantendrá persistente durante la vida completa del componente.



reglas de los hooks



Los hooks son funciones de JavaScript, pero necesitas seguir dos reglas cuando los uses.

- Llama a los Hooks solo en el nivel superior 

→ No llames Hooks dentro de ciclos, condicionales o funciones anidadas.

Siguiendo esta regla, te asegurar de que los hooks se llamen en el mismo orden cada vez que un componente se renderiza.

- Llama Hooks solo en funciones de React. 

→ No llames Hooks desde funciones JavaScript regulares.

Siguiendo esta regla, te asegurar de que toda la lógica del Estado de un componente sea claramente visible desde tu código fuente.

Hay un plugin de ESLint llamado eslint-plugin-react-hooks que refuerza estas dos reglas.
Este plugin es incluido por defecto en Create React App.